命名空间和模块

一、命名空间

1.1 单文件命名空间

```
namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }

    const lettersRegexp = /^[A-Za-z]+$/;
    const numberRegexp = /^[0-9]+$/;

    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }

    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}

let strings = ["hello", "98025", "101"];
let validators: { [s: string]: Validation.StringValidator; } = {};
validators['ZIP CODE'] = new Validation.ZipCodeValidator();
validators['Letters only'] = new Validation.LettersOnlyValidator();

for (let s of strings) {
    for (let name in validators) {
        console.log(`"${s}" - ${validators[name].isAcceptable(s) ? "matches" : "does not match"}
    }
}
```

1.2 多文件命名空间

应用变得越来越大，代码一般会分到不同的文件中，便于维护，于是有了多文件命名空间。尽管文件不同，但是属于同一个命名空间，使用的时候像在一个文件中定义的一样。不同的文件之间存在依赖关系，所以可以使用**引用标签**来告诉编译器之间的关联。

```ts
//Validation.ts
namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }
}

//LettersOnlyValidator.ts
///<reference path="Validation.ts"/>

namespace Validation {
    const lettersRegexp = /^[A-Za-z]+$/;
    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }

}
//ZipCodeValidator.ts
///<reference path="Validation.ts"/>
namespace Validation {
    const numberRegexp = /^[0-9]+$/;
    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}

//test.ts
///<reference path="Validation.ts"/>
///<reference path="ZipCodeValidator.ts"/>
///<reference path="LettersOnlyValidator.ts"/>

let strings = ["hello", "98025", "101"];
let validators: { [s: string]: Validation.StringValidator; } = {};
validators['ZIP CODE'] = new Validation.ZipCodeValidator();
validators['Letters only'] = new Validation.LettersOnlyValidator();

for (let s of strings) {
    for (let name in validators) {
        console.log(`"${s}" - ${validators[name].isAcceptable(s) ? "matches" : "does not match"}
    }
}
```

涉及多文件的时候，必须要确保所有编译后的代码都被加载了，加载代码有两种方式：

- 所有输入文件编译为一个输出文件，需要使用--outFile标记： tsc --outFile sample.js Test.ts。编译器会根据源码里的引用标签自动对输出排序，也可以单独指定每个文件：

```
tsc --outFile sample.js Validation.ts LettersOnlyValidator.ts ZipCodeValidator.ts Test.ts
```

- 可以编译每一个文件，每个源文件都会对应生成一个js文件，然后在页面上用script标签把所有生成的js文件，按照正确的顺序加载进来。

1.3 别名

别名可以简化命名空间，使用:

```
import q = x.y.z;
```

不要与用来加载模块的:

```
import x = require(`name`);
```

语法混淆，这里的语法是为指定的符号创建一个别名，可以用这种方法为任意的标识符创建别名，包括导入的对象。

```
namespace Shapes {
    export namespace Polygons {
        export class Triangle {

        }

        export class Square {

        }
    }
}

import polygons = Shapes.Polygons;

let sq = new polygons.Square();
```

暂时没有使用require，直接导入符号的限定名赋值，这与var类似，但还适用于类型和导入的具体命名空间含义的符号，重要的是对值来讲，import会生成和原始符号不同的引用，所以改变别名 的var值并不会影响原始变量的值。

1.5 外部命名空间

declare。
D3库的源码：

```
declare namespace D3{
    export interface Selectors{
        select: {
            (selector: string): Selection;
            (element: EventTarget): Selection;
        }
    }

    export interface Event{
        x: number;
        y: number;
    }

    export interface Base extends Selectors{
        event: Event;
    }
}

declare var d3: D3.Base;
```

二、模块

ES6开始引入了模块的概念。
模块在自身的作用于里执行，而不是在全局作用域里。以为这定义在一个模块里的变量、函数、类等在模块外部是不可见的，除非你明确export导出它。相反，如果想使用其他的模块导出的变量、函数、类、接口等，必须要导入他们，可以使用import。

模块是自声明的，两个模块之间的关系，通过文件级别上使用import和export建立的。

使用模块加载器去导入其他模块，模块加载器的作用是在执行模块代码之前去查找并执行这个模块的所有依赖。

TS和ES6一样，任何包含顶级import或者export的文件都被当成一个模块。如果一个文件不带有顶级的import后者export声明，那么他的内容被视为全局可见，对模块也是可见的。

2.1 导出

2.1.1 导出声明

任何声明都可以通过export关键字导出。

2.1.2 导出语句

导出语句很方便，我们可能需要对导出的部分重命名。

```
class A{

}
export { A}
export { A as newA}
```

2.1.3 重新导出

我们经常会扩展其他模块，只导出那个模块的部分内容，重新导出功能并不会在当前模块导入那个模块或者定义一个新的局部变量。

比如原来ZipCodeValidator.ts里，有导出一个ZipCodeValidator。
在新的ParseIntBasedZipCodeValidator.ts中：

```
export {ZipCodeValidator as RegZipCodeValidator} from "./ZipCodeValidator";
```

或者一个模块可以包裹其它的多个模块，将其导出的内容联合在一起，使用语法:

> export * from "";

2.2 导入

2.2.1 导入一个模块中的导出内容

```
import { A} from "module";
```

2.2.2 对导入内容重命名

```
import {A as newA} from "module";
```

也可以将整个模块导入到一个变量，并通过它访问模块的导出部分。

```
import * as validator from "";

let myValidator = new validator.ZipCodeValidator();
```

2.2.3 具有副作用的导入

尽管不推荐这么做，一些模块会设置一些全局状态供其他的模块使用，这些模块可能没有任何导出或者用户根本不关心它的导出。

```
import "./module.js";
```

2.3 默认导出

每个模块都可以有一个默认导出，默认导出使用default关键字，一个模块只能有一个默认导出。需要一个特殊的形式来导入默认导出。
类和函数声明可以直接报标记为默认导出、标记为默认导出的类和函数的名字是可以省略的。所以再导入的时候可以重新赋值，名字可以随便定。
如:

```
//A.ts
 export default class A{

 }

//B.ts
import newA from "./A";
```

2.3.1 export 和import = require()

export = 语法定义一个模块的导出对象，可以是类、接口、命名空间、函数和枚举。
如果导入一个使用了export = 的模块时，必须使用Ts特定于法:

> import module = require("module");

```
//ZipCodeValidator.ts
let numberRegexp = /^[0-9]+$/;
class ZipCodeValidator {

}

export = ZipCodeValidator;

//Test.ts
import zip = require("./ZipCodeValidator");

let validator = new zip();
```

2.4 生成模块

2.5 外部模块

下面代码只演示用法，并未准确表示。

```
declare module "url"{
    export interface Url{

    }
    export function parse(){

    }
}

declare module "path"{
    export function a();
    export function b();
    export let sep: string;
}

import * as URL from  "url";
let myUrl = URL.parse();
```

2.5.1 简写

假如不想在使用一个模块之前花时间去写声明，可以采用简写声明。

```
declare module "hot-new-module";
```

简写里所有导出的类型都是any

2.5.2 模块声明通配符

某些模块加载器支持导入非js的内容，通常会使用一个前缀或者后缀来标识特殊的加载语法。

```
declare module "*!text"{
    const content: string;
    export default content;
}

declare module "json!*"{
    const value: any;
    export default value;
}
```

现在可以导入匹配*!text或者json!*的内容了。

```
import fileContent from "./xyz.txt!text";

import data from "json!http://www.example.com/data.json";

console.log(data,fileContent);
```

2.6 UMD模块

有些模块能兼容多个模块加载器，或者不适用模块加载器，它们以UMD为代表。可以通过导入的形式或者全局变量的方式访问。

```
export function isPrime(x: number): boolean{

}

export as namespace mathLib;
```

之后这个库可以在某个模块里导入来使用：

```
import {isPrime} from "math-lib";

isPrime(2);

mathLib.isPrime(2);//错误，can not use the global definition from inside module.
```

它同样可以 通过全局变量的形式使用，但只能在某个脚本里用（脚本是指没有导入或者导出的文件）。